

# Probabilistic Resource Analysis by Program Transformation

Maja H. Kirkeby and Mads Rosendahl

Computer Science, Roskilde University  
Roskilde, Denmark  
`majaht@ruc.dk`, `madsr@ruc.dk` \*\*

**Abstract.** The aim of a probabilistic resource analysis is to derive a probability distribution of possible resource usage for a program from a probability distribution of its input. We present an automated multi-phase rewriting based method to analyze programs written in a subset of C. It generates a probability distribution of the resource usage as a possibly uncomputable expression and then transforms it into a closed form expression using over-approximations. We present the technique, outline the implementation and show results from experiments with the system.

## 1 Introduction

The main contribution in this paper is to present a technique for probabilistic resource analysis where the analysis is seen as a program-to-program translation. This means that the transformation to closed form is a source code program transformation problem and not specific to the analysis. Any necessary approximations in the analysis are performed at the source code level. The technique also makes it possible to balance the precision of the analysis against the brevity of the result.

Many optimizations for increased energy efficiency require probabilistic and average-case analysis as part of the transformations. Wierman et al. state that “*global energy consumption is affected by the average-case, rather than the worst case*” [36]. Also in scheduling “*an accurate measurement of a task’s average-case execution time can assist in the calculation of more appropriate deadlines*” [17]. For a subset of programs a precise average-case execution time can be found using static analysis [12, 30, 14]. Applications of such analysis may be in improving scheduling of operations or in temperature management. Because the analysis returns a distribution, it can be used to calculate the probability of energy consumptions above a certain limit, and thereby indicate the risk of over-heating.

---

\*\* The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRa - Whole-Systems Energy Transparency.

The central idea in this paper is to use probabilistic output analysis in combination with a preprocessing phase that instruments programs with resource usage. We translate programs into an intermediate language program that computes the probability distribution of resource usage. This program is then analyzed, transformed, and approximated with the aim of obtaining a closed form expression. It is an alternative to deriving cost relations directly from the program [7] or expressing costs as abstract values in a semantics for the language.

As with automatic complexity analysis, the aim of probabilistic resource analysis is to express the result as a parameterized expression. The time complexity of a program should be expressed as a closed form expression in the input size, and for probabilistic resource analysis, the aim is to express the probability of resource usage of the program parameterized by input size or range. If input values are not independent, we can specify a joint distribution for the values. Values do not have to be restricted to a finite range but for infinite ranges the distribution would converge to zero towards the limit.

The current work extends our previous work on probabilistic analysis [28] in three ways. We show how to use a preprocessing phase to instrument programs with resource usage such that the resource analysis can be expressed as an analysis of the possible output of a program. The resource analysis can handle an extended class of programs with structured data as long as the program flow does not depend on the probabilistic data in composite data structures. Finally, we present an implementation of the analysis in the Ciao language [5] which uses algebraic reductions in the Mathematica system [38].

The focus in this paper is on using fairly simple local resource measures where we count core operations on data. Since the instrumentation is done at the source code level, we can use flow information so that the local costs can depend on actual data to operations and which operations are executed before and after. This is normally not relevant for time complexity but does play an important role for energy consumption analysis [19, 31].

## 2 Probability distributions in static analysis

In our approach to probabilistic analysis, the result of an analysis is an approximation of a probability distribution. We will here present the concepts and notation we will use in the rest of the paper. A probability distribution is also often referred to as the *probability mass function* in the discrete case, and in the continuous case, it is a *probability density functions*. We will use an upper case  $P$  letter to denote a probability distribution.

**Definition 1 (input probability).** *For a countable set  $X$  an input probability distribution is a mapping  $P_X : X \rightarrow \{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$ , where*

$$\sum_{x \in X} P_X(x) = 1$$

We define the output probability distribution for a program  $p$  in a forward manner. It is the *weight* or sum of all probabilities of input values where the program returns the desired value  $z$  as output.

**Definition 2 (output probability).** *Given a program,  $p : X \rightarrow Z$  and a probability distribution for the input,  $P_X$ , the output probability distribution,  $P_p(z)$ , is defined as:*

$$P_p(z) = \sum_{x \in X \wedge p(x)=z} P_X(x)$$

Note that Kozen also uses a similar forward definition [20], whereas Monniaux constructs the inverse mapping from output to input for each program statement and expresses the relationship in a backwards style [23].

**Lemma 1.** *The output probability distribution,  $P_p(z)$ , satisfies*

$$0 \leq \sum_z P_p(z) \leq 1$$

The program may not terminate for all input, and this means that the sum may be less than one. If we expand the domain  $Z$  with an element to denote non-termination,  $Z_\perp$ , the total sum of the output distribution  $P_p(z)$  would be 1.

In our static analysis, we will use approximations to obtain safe and simplified results. Various approaches to approximations of probability distributions have been proposed and can be interpreted as *imprecise probabilities* [1, 10, 9]. Dempster-Shafer structures [16, 2] and P-boxes [11] can be used to capture and propagate uncertainties of probability distributions. There are several results on extending arithmetic operations to probability distributions for both known dependencies between random variables and when the dependency is unknown or only partially known [3, 4, 18, 32, 37]. Algorithms for lifting basic operations on numbers to basic operations on probability distributions can be used as abstractions in static analysis based on abstract interpretation. Our approach uses the P-boxes as bounds of probability distributions. P-boxes are normally expressed in terms of the cumulative probability distribution but we will here use the probability mass function. We do not, however, use the various basic operations on P-boxes, but apply approximations to a probability program such that it forms a P-box.

**Definition 3 (over-approximation).** *For a distribution  $P_p$  an over-approximation ( $\bar{P}_p$ ) of the distribution satisfies the condition:*

$$\bar{P}_p : \forall z. P_p(z) \leq \bar{P}_p(z) \leq 1 .$$

The aim of the probabilistic resource analysis is to derive an approximation  $\bar{P}_p$  as tight as possible.

The over-approximation of the probability distribution can be used to derive lower and upper bounds of the expected value and will thus approximate the expected value as an interval [28].

### 3 Architecture of the transformation system

The system contains five main phases. The input to the system is a program in a small subset of C with annotations of which part we want to analyze. It could be the whole program but can also be a specific subroutine which is called repeatedly with varying arguments according to some input distribution.

The first phase will instrument the program with resource measuring operations. The instrumented program will perform the same operations as the original program in addition to recording and printing resource usage information. This program can still be compiled and run, and it will also produce the same results as the original program.

The second phase translates the program into an intermediate language for further analysis. We use a small first-order functional language for the analysis process. The translation has two core elements. We slice [35] the program with respect to the resource measuring operations and transform loops into primitive recursion in the intermediate language. The transformed program can still be executed and will produce the same resource usage information as the instrumented program. Since the instrumentation is done before the translation to intermediate language any interpretation overhead or speed-up due to slicing does not influence the result [27].

In the third phase, we construct a probability output program that computes the probability output function. In this case, it is a probability distribution of possible resource usages of the original program. This program can also run but will often be extremely inefficient since it will merge information for all possible input to the original program.

The fourth phase transforms the probability program into a large expression without further function calls. Recursive calls are removed using summations and the transformed program computes the same result as the program did before this phase.

In the final phase the probability function is transformed into closed form using symbolic summation and over-approximation. In this phase we exploit the Mathematica system [38]. The final probability program computes the same result or an over-approximation of the function produced in the fourth phase.

### 4 Instrumenting programs for resource analysis

The input to the analysis is a program in a subset of C. In the next section we define the intermediate language for further analysis and it is the restrictions on the intermediate language that limits the source programs we can analyze with our system. The source program may contain integer variable and arrays, usual loop constructs and non-recursive function calls. The program should be annotated with specification on which part of the program to analyze. The following is an example of such a program.

```
// Toanalyze: multa(_,_,,N)
```

```

void multa(int a1[MX],int a2[MX],int a3[MX],int n){
    int i1,i2,i3,d;
    for(i1 = 0; i1 < n; i1++) {
        for(i2 = 0; i2 < n; i2++) {
            d = 0;
            for(i3 = 0; i3 < n; i3++) {
                d = d + a1[i1*n+i3]*a2[i3*n+i2];
            }
            a3[i1*n+i2] = d;
        }
    }
}

```

This example program describes a matrix multiplication for which we would like to analyze the probability distribution for the number of steps when parameterized with the size (N) of the matrices.

**Instrumentation.** The program is then instrumented with resource usage information and translated into an intermediate language for further analysis. The instrumented program is also a valid program in the source language and can be executed with the same results as the original program. It will, however, also collect resource usage information.

In our example, we instrument the program with step counting information where we count the number of assignment statement being executed. This is done by inserting a variable into the program and incrementing it once for each assignment statement.

```

int multa(int a1[MX],int a2[MX],int a3[MX],int n){
    int i1,i2,i3,d;
    int step; step=0;
    for(i1 = 0; i1 < n; i1++) {
        for(i2 = 0; i2 < n; i2++) {
            d = 0; step++;
            for(i3 = 0; i3 < n; i3++) {
                d = d + a1[i1*n+i3]*a2[i3*n+i2]; step++;
            }
            a3[i1*n+i2] = d; step++;
        }
    }
    return step;
}

```

The outer loop does not update the step counter, whereas the first inner loop updates it twice per iteration and the innermost loop updates it once per loop iteration.

**Slicing.** The second phase will slice the program with respect to resource usage and translate the program into the intermediate language of first-order functions that we will use in the subsequent stages. Loops in the program are translated into primitive recursion.

```

for3(i3, step, n) =
  if(i3 = n) then step else for3(i3 + 1, step+1, n)

for2(i2, step, n) =
  if(i2 = n) then step else for2(i2 + 1, for3(0, step+2, n), n)

for1(i1, step, n) =
  if(i1 = n) then step else for1(i1 + 1, for2(0, step, n), n)

tmulta(n) = for1(0, step, n)

```

Each function in the recursive program corresponds to a for loop with their related step-updates. The step counter is given as input argument to the next function in a continuation-passing style.

**Intermediate language.** An intermediate program,  $\text{Prg}$ , consists of integer functions,  $f_i: \text{Int}^* \rightarrow \text{Int}$ , as given by the abstract syntax given in Figure 1. In the following, we relax the restrictions on function and parameter names.

$$\begin{aligned}
f_i(x_1, \dots, x_n) &\stackrel{\text{def}}{=} \langle \text{exp} \rangle \\
\langle \text{aexp} \rangle &|= \mathbf{x}_i \mid \mathbf{c} \mid \langle \text{aexp} \rangle +_i \langle \text{aexp} \rangle \mid \langle \text{aexp} \rangle -_i \langle \text{aexp} \rangle \mid \\
&\quad \langle \text{aexp} \rangle \times_i \langle \text{aexp} \rangle \mid \langle \text{aexp} \rangle \text{div}_i \langle \text{aexp} \rangle \\
\langle \text{bexp} \rangle &|= \langle \text{aexp} \rangle =_i \langle \text{aexp} \rangle \mid \langle \text{aexp} \rangle <_i \langle \text{aexp} \rangle \mid \langle \text{aexp} \rangle \leq_i \langle \text{aexp} \rangle \mid \\
&\quad \mathbf{true} \mid \mathbf{false} \mid \mathbf{not}(\langle \text{bexp} \rangle) \\
\langle \text{exp} \rangle &|= \langle \text{aexp} \rangle \mid f_i(\langle \text{exp}_1 \rangle, \dots, \langle \text{exp}_n \rangle) \mid \\
&\quad \mathbf{if} \langle \text{bexp} \rangle \mathbf{then} \langle \text{exp} \rangle \mathbf{else} \langle \text{exp} \rangle
\end{aligned}$$

**Fig. 1.** The abstract syntax describing the intermediate programs.

**Definition 4.** A program is well-formed if it follows the abstract syntax and it contains a finite number of function definitions, that each is of one of the following forms and can internally be enumerated with a natural number such that:

$$\begin{aligned}
f_i(x_1, \dots, x_n) &\stackrel{\text{def}}{=} \mathbf{if} \ b \ \mathbf{then} \ e_0 \ \mathbf{else} \ f_i(e_1, \dots, e_n) \\
&\text{where } f_i \text{ is simple, } e_0 \text{ only contains calls to functions } f_j \text{ where } j < i.
\end{aligned}$$

$$\begin{aligned}
f_i(x_1, \dots, x_n) &\stackrel{\text{def}}{=} e \\
&\text{where } e \text{ only contain calls to functions } f_j \text{ where } j < i.
\end{aligned}$$

The enumeration prevents mutual recursion and ensures that non-recursive calls cannot create an infinite call-chain.

## 5 Probabilistic output analysis

The analysis is applied to the intermediate program and an input probability program in the intermediate language. The output is a new program that can be

described by a subset of the intermediate language; this will be clarified later in the definition of pure and closed form programs. The analysis consists of three phases:

- Create, where the probability program describing the output distribution is created as a possibly uncomputable expression.
- Separate, where we remove all calls from the probability program.
- Simplify, where we transform the program into closed form using safe over-approximations when necessary.

The analysis is constructed as three sets of transformation rules, one for each of the three phases. All transformations are syntax directed, and a strategy is to apply them in a depth-first manner. The program output analysis is implemented in Ciao and integrates with Mathematica in the third phase to reduce expressions.

In the following we use  $\mathcal{V}ar(e)$  to represent the set of variables occurring in expression  $e$ , and  $\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \stackrel{\text{def}}{=} e$  to represent the function  $\mathbf{f}$  is defined in the input program. Some side conditions are explained in an informal way, as in “ $\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \stackrel{\text{def}}{=} e$ , where  $e$  is non-recursive”.

$$\text{name} \frac{\text{precondition}_1 \quad \dots \quad \text{precondition}_n}{\text{original term} \rightarrow \text{rewritten term}}$$

The preconditions are evaluated from left to right, and if all succeeds, we can use the transformation. When substituting a variable  $\mathbf{x}$  to an expression  $e$ , we denote it  $[\mathbf{x}/e]$ .

In the following we will begin by extending the intermediate language presented in Figure 1 such that it can express probabilities, and afterwards describe the transformation rules for each phase.

**The intermediate language.** The intermediate language is, as previously mentioned, a first-order functional language. A probability program can be evaluated at any stage through the transformation process.

We extend the abstract syntax given in Figure 1 such that it can easily describe probability distributions. We introduce probability functions,  $\mathbf{P}: \text{Int}^* \rightarrow \text{Real}$ , which follows the expanded syntax given in Figure 2. The dots indicate the syntax described in Figure 1. Again,  $\langle \text{aexp} \rangle$  and  $\langle \text{exp} \rangle$  are of type integer,  $\langle \text{bexp} \rangle$  is boolean, and the new  $\langle \text{qexp} \rangle$  is a real. In  $\langle \text{qexp} \rangle$  the method `i2r` type casts an integer expression to a real. We introduce `c`, `sum`, `prod` and `argDev` functions. `c` evaluates to either 1, if its boolean expression evaluates to true, or 0 when it evaluates to false. Evaluating `sum` instantiates the variable with all possible values and sum all the results of the evaluation the  $\langle \text{qexp} \rangle$ . `prod` instantiates its variable with all values for which the first  $\langle \text{qexp} \rangle$  evaluates to 1, and then it multiply all the results from evaluating the second  $\langle \text{qexp} \rangle$ . The last expression introduced is `argDev` which describes the development of the variable  $\mathbf{x}_i$  as a function of the number of updates,  $\mathbf{x}_j$ . The expression  $\langle \text{exp} \rangle$  computes the development of  $\mathbf{x}_i$  for one incrementation of  $\mathbf{x}_j$  (e.g. the argument

$x_i$  in a function  $f(x_i)$  with a recursive call  $f(x_i-1)$  has a argument development  $\text{argDev}(x_i, x_i-2, x_j)$ .

$$\begin{aligned}
f_i(x_1, \dots, x_n) &\stackrel{\text{def}}{=} \langle \text{exp} \rangle \\
\langle \text{aexp} \rangle &\models \dots \mid \min(\langle \text{aexp} \rangle, \langle \text{aexp} \rangle) \mid \max(\langle \text{aexp} \rangle, \langle \text{aexp} \rangle) \\
\langle \text{bexp} \rangle &\models \dots \mid \langle \text{aexp} \rangle =_i \langle \text{exp} \rangle \\
\langle \text{exp} \rangle &\models \dots \mid \text{argDev}(x_i, \langle \text{exp} \rangle, x_j) \\
P_i(x_1, \dots, x_n) &\stackrel{\text{def}}{=} \langle \text{qexp} \rangle \\
\langle \text{qexp} \rangle &\models \text{i2r}(\langle \text{aexp} \rangle) \mid \text{c}(\langle \text{bexp} \rangle) \mid \langle \text{qexp} \rangle \text{op}_q \langle \text{qexp} \rangle \mid \\
&\quad \text{sum}(x_i, \langle \text{qexp} \rangle) \mid \text{prod}(x_i, \langle \text{qexp} \rangle, \langle \text{qexp} \rangle) \mid \\
&\quad P_i(\langle \text{aexp}_1 \rangle, \dots, \langle \text{aexp}_n \rangle) \\
\text{op}_q &= +_q \mid -_q \mid \times_q \mid /_q
\end{aligned}$$

**Fig. 2.** The expanded abstract syntax describing probability programs.

A program that computes a probability distribution is referred to as a probability program.

**Definition 5.** A probability program that has no if-expressions no function calls is pure and a pure probability program without any *sum* and *prod* is in closed form.

A program is *pure* after it is transformed in the separation phase and is pure and in *closed form* after the simplification phase.

**The create phase.** This phase has only one rule which creates a program that computes a probability distribution from the intermediate program and input distributions.

$$\text{create} \frac{f(u_1, \dots, u_n) \stackrel{\text{def}}{=} e \quad P(v_1, \dots, v_n) \stackrel{\text{def}}{=} e_p}{P_f(z) \stackrel{\text{def}}{=} \text{sum}(x_1; \dots \text{sum}(x_n; \text{c}(z =_i f(x_1, \dots, x_n)) \times_q P(x_1, \dots, x_n)))}$$

We use the create rule to make a new probability function describing the probability distribution for the integer function we are interested in.

**The separate phase.** In this phase function calls are removed by repeatedly exposing calls and replacing them. Non-recursive function calls are unfolded using their definitions. Function calls can occur inside if-expressions or as nested



calls; these are extracted and handled one at a time.

$$\begin{array}{l}
\text{f-simple} \frac{\mathbf{f}(y_1, \dots, y_n) \stackrel{\text{def}}{=} e \quad , \text{ where } e \text{ is non-recursive} \quad \mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{Var}}{\mathbf{c}(\mathbf{z} =_i \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n)) \rightarrow \mathbf{c}(\mathbf{z} =_i e[\mathbf{y}_1/\mathbf{x}_1, \dots, \mathbf{y}_n/\mathbf{x}_n])} \\
\\
\text{rem-P} \frac{\mathbf{P}(x_1, \dots, x_n) \stackrel{\text{def}}{=} e}{\mathbf{P}(e_1, \dots, e_n) \rightarrow e[x_1/e_1, \dots, x_n/e_n]} \\
\\
\text{rem-if} \frac{}{\mathbf{c}(\mathbf{z} =_i \text{if } b \text{ then } e_0 \text{ else } e_1) \rightarrow (\mathbf{c}(b) \times_q \mathbf{c}(\mathbf{z} =_i e_0) +_q \mathbf{c}(\text{not}(b)) \times_q \mathbf{c}(\mathbf{z} =_i e_1))} \\
\\
\text{no-nest}(\mathbf{f}) \frac{\{e_1, \dots, e_n\} \not\subseteq \mathcal{Var}}{\begin{array}{l} \mathbf{c}(\mathbf{z} =_i \mathbf{f}(e_1, \dots, e_n)) \rightarrow \\ \mathbf{sum}(\mathbf{u}_1; \dots \mathbf{sum}(\mathbf{u}_n; \mathbf{c}(\mathbf{z} =_i \mathbf{f}(\mathbf{u}_1, \dots, \mathbf{u}_n)) \times_q \mathbf{c}(\mathbf{u}_1 =_i e_1) \times_q \dots \times_q \mathbf{c}(\mathbf{u}_n =_i e_n))) \end{array}}
\end{array}$$

We replace calls to recursive functions by a summation over the number of recursions using argument development constructors to describe the value of each argument as a function of the index of the summation. This way of defining argument development has similarities with size change functions derived using recurrence equations. Argument development functions do not depend on the base-case unlike size-change functions [39]. The summation also contains a product which ensures that the condition evaluates to false for argument values less than the current value of the index of summation. When the expression in a product contains only c-constructors, then the product is evaluated to 1 if either the range is empty or the expression is evaluated to true for the full range. The following rewrite rules are all that is needed for transforming probability programs into pure probability programs.

$$\begin{array}{l}
\text{f-rec} \frac{\mathbf{f}(y_1, \dots, y_n) \stackrel{\text{def}}{=} \text{if } b \text{ then } e_0 \text{ else } \mathbf{f}(e_1, \dots, e_n) \quad \mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{Vars}}{\begin{array}{l} \sigma_{y/i} = [\mathbf{y}_1/\mathbf{i}_1, \dots, \mathbf{y}_n/\mathbf{i}_n] \sigma_{y/x} = [\mathbf{y}_1/\mathbf{x}_1, \dots, \mathbf{y}_n/\mathbf{x}_n] \sigma_{y/j} = [\mathbf{y}_1/\mathbf{j}_1, \dots, \mathbf{y}_n/\mathbf{j}_n] \\ \mathbf{c}(\mathbf{z} =_i \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n)) \rightarrow \\ \mathbf{sum}(\mathbf{i}; \mathbf{c}(0 \leq_i \mathbf{i}) \times_q \\ \mathbf{sum}(\mathbf{i}_1; \dots \mathbf{sum}(\mathbf{i}_n; \mathbf{c}(\sigma_{y/i}(b)) \times_q \mathbf{c}(\mathbf{i}_1 =_i \mathbf{argDev}(\mathbf{x}_1, \sigma_{y/x}(e_1), \mathbf{i})) \times_q \\ \mathbf{c}(\mathbf{z} =_i \sigma_{y/i}(e_0)) \times_q \dots \times_q \mathbf{c}(\mathbf{i}_n =_i \mathbf{argDev}(\mathbf{x}_n, \sigma_{y/x}(e_n), \mathbf{i})) \dots) \times_q \\ \mathbf{prod}(\mathbf{j}; \mathbf{c}(0 \leq_i \mathbf{j}) \times_q \mathbf{c}(\mathbf{j} \leq_i \mathbf{i}-1); \\ \mathbf{sum}(\mathbf{j}_1; \dots \mathbf{sum}(\mathbf{j}_n; \mathbf{c}(\text{not}(\sigma_{y/j}(b))) \times_q \\ \mathbf{c}(\mathbf{j}_1 =_i \mathbf{argDev}(\mathbf{x}_1, \sigma_{y/x}(e_1), \mathbf{j})) \times_q \dots \times_q \mathbf{c}(\mathbf{j}_n =_i \mathbf{argDev}(\mathbf{x}_n, \sigma_{y/x}(e_n), \mathbf{j})) \\ \dots))) \end{array}}
\end{array}$$

The argument development expression may contain function calls as well, and these are extracted equivalently to nested functions.

$$\text{no-nest}(\mathbf{argDev}) \frac{}{\begin{array}{l} \mathbf{c}(\mathbf{z} =_i \mathbf{argDev}(\mathbf{x}, \mathbf{f}(e_1, \dots, e_n), \mathbf{i})) \rightarrow \\ \mathbf{sum}(\mathbf{u}; \mathbf{c}(\mathbf{z} =_i \mathbf{argDev}(\mathbf{x}, \mathbf{f}(e_1, \dots, e_n), \mathbf{i})) \times_q \mathbf{c}(\mathbf{u} =_i \mathbf{f}(e_1, \dots, e_n))) \end{array}}$$

After applying these rules until they cannot be applied anymore, the probability program has been transformed to pure form.

**The simplification phase.** We have presented the rules for obtaining a pure probability program, and in this section we outline the rules used to reach closed

form. A pure probability function consists of a series of nested summations multiplied with an expression (e.g. input probability). The rules are applied in no particular order and the phase ends when no more rules can be applied. In this phase we integrate with Mathematica. A call to Mathematica is denoted  $\text{mm:Function}(Arg) = Answer$ , where **Function** denotes the actual function called in Mathematica (e.g. **mmExpand** calls Mathematica's **Expand** function). The translation between the intermediate language and Mathematica's representation will not be discussed further here.

The rules can be grouped by their functionality: preparing expressions, removal of summations and removal of products. The latter are currently the only rules containing over-approximations.

**Preparing** expressions for removal of either summations or products involve moving expressions that do not depend on the index of summation outside the summation, dividing summations of additions into simpler ones, reducing expressions, dividing summations in ranges, and remove argument development constructors. Please notice that  $\text{div-sum}(x \leq)$  has an equivalent rule for upper bounds.

$$\begin{array}{l}
\text{move-c} \frac{x \notin \text{Var}(e_1)}{\text{sum}(x; e_1 \times_q e_2) \rightarrow e_1 \times_q \text{sum}(x; e_2)} \\
\\
\text{div-sum}(+) \frac{x \in \text{Var}(e_1) \quad x \in \text{Var}(e_2)}{\text{sum}(x; e_1 +_q e_2) \rightarrow \text{sum}(x; e_1) +_q \text{sum}(x; e_2)} \\
\\
\text{div-sum}(x \leq) \frac{x \notin \text{Var}(e_1, e_2) \quad x \in \text{Var}(e_2)}{\begin{array}{l} \text{sum}(x; c(x \leq_i e_1) \times_q c(x \leq_i e_2) \times_q e_3) \rightarrow \\ c(e_1 \leq_i e_2) \times_q \text{sum}(x; c(x \leq_i e_1) \times_q e_3) +_q \\ c(e_2 \leq_i e_1 -_i 1) \times_q \text{sum}(x; c(x \leq_i e_2) \times_q e_3) \end{array}} \\
\\
\text{rem(argDev)} \frac{c \in \mathbf{n}}{c(z =_i \text{argDev}(x, x +_i c, i)) \rightarrow c(z =_i x +_i c \times_i i)} \\
\\
\text{reduceAexp} \frac{\text{mm:Reduce}(e_1) = e_2}{c(e_1) \rightarrow c(e_2)} \\
\\
\text{reduce}(=) \frac{}{c(\text{true}) \rightarrow \text{i2r}(1)}
\end{array}$$

**Removal of summations** can be done in two ways. Either the index of the summation can only be one value or it can be a limited range of values, and depending on which case different transformations are used. In the first case, there exists an equation containing the variable index of the innermost summation. The equation is solved for the variable, and the rest of the variable occurrences are replaced by the new value.

$$\text{rem-sum}(=) \frac{\text{mm:Solve}(e_1 =_i e_2, x) = c(x =_i e_3)}{\text{sum}(x; c(e_1 =_i e_2) \times_q e) \rightarrow e[x/e_3]}$$

Removing a summation by its range involves using standard mathematical formulas for rewriting series. The last part of the following rule uses  $\sum_{k=1}^n k^2 = n(n+1)(2n+1)/6$ . We only present transformations up to quadratic series and our pragmatic implementation contains rules for transforming series of power of degree up to 10. A more general rewrite rule for series of power of degree up to  $p$  could be implemented, but is more complicated as it includes Bernoulli numbers and binomial coefficients. The precondition uses Mathematica's `Expand` to transform the expression into the right pattern.

$$\text{rem-sum}(\leq) \frac{x \notin \mathcal{Var}(e_1, \dots, e_6) \quad \text{mm:Expand}(e_3) = \text{i2r}(e_4 +_i e_5 \times_i x +_i e_6 \times_i x \times_i x)}{\begin{array}{l} \text{sum}(x; c(e_1 \leq_i x) \times_q c(x \leq_i e_2) \times_q \text{i2r}(e_3)) \rightarrow \\ \text{i2r}(e_4) \times_q \text{i2r}(e_2 -_i e_1 +_i 1) +_q \\ \text{i2r}(e_5) \times_q \text{i2r}(e_2 \times_i (e_2 +_i 1)) /_q 2 -_q \\ \text{i2r}(e_5) \times_q \text{i2r}(e_2 \times_i (e_2 -_i 1)) /_q 2 +_q \\ \text{i2r}(e_6) \times_q \text{i2r}(e_2 \times_i (e_2 +_i 1) \times_i (2 \times_i e_2 +_i 1)) /_q 6 -_q \\ \text{i2r}(e_6) \times_q \text{i2r}(e_2 \times_i (e_2 -_i 1) \times_i (2 \times_i e_2 -_i 1)) /_q 6 \end{array}}$$

**Removal of Product** involves a safe over-approximation. The implementation of POA contains two different over-approximations and in many cases the probability program can be transformed into closed form in a precise manner. In the following paragraph we describe when the transformation preserves the accuracy of the transformed term.

The probability function can always be over-approximated to 1. The rule `f-rec` is an exact rule and introduces a product-expression which may not be possible to rewrite into closed form. We only introduce the product-expression with `c`-expressions in its body, and therefore it will always either evaluate to 1 or 0. A safe over-approximation of such a product-expression is 1.

$$\text{rem-prod-one} \frac{x \notin \mathcal{Var}(e_1, e_2) \quad x \in \mathcal{Var}(e_3)}{\text{prod}(x; c(e_1 \leq_i x) \times_q c(x \leq_i e_2); c(e_3)) \rightarrow 1}$$

For the summation describing recursive calls, this transformation is exact when the condition,  $b$ , evaluates to true for exactly one value (eg. it is an equation).

A broader class of recursive programs (than those having an equation in the condition) is those where the `c`-expression is monotone in  $x$ ; meaning that there exists a  $k$  for which  $c(e_3) = 1$  for  $x \leq k$  and  $c(e_3) = 0$  for  $x > k$ . This case covers many for-loops. In this case, we can accurately replace the `prod`-expression with two `c`-expressions one checking the lower and one checking the upper range-limit. The empty product (the lower limit is larger than the upper) is 1.

$$\text{rem-prod-mon} \frac{x \notin \mathcal{Var}(e_1, e_2) \quad x \in \mathcal{Var}(e_3) \quad e_3 \text{ is monotone in } x}{\begin{array}{l} \text{prod}(x; c(e_1 \leq_i x) \times_q c(x \leq_i e_2); c(e_3)) \rightarrow \\ (c(e_3)[x/e_1] \times_q c(e_3)[x/e_2] \times_q c(e_1 \leq_i e_2) +_q c(e_2 \leq_i e_1 -_i 1)) \end{array}}$$

This rule does not preserve accuracy when the `c`-expression is not monotone in  $x$  (e.g.  $c(2 \leq_i x || 4 \leq_i x)$ ).

## 6 Implementation and results

In the following, we present three examples which show results of programs with nested loops parameterized input distribution of multiple variables. The probability distribution computed by the output program varies in complexity; the first program calculates a single parameterized output, the second program computes a triangular shaped output distribution and third computes a distribution converging towards a standard normal distribution. The results are presented in a reduced and readable form extracted from our implementation.

**Matrix multiplication.** The original matrix multiplication program uses composite types and contains nested loops. The intermediate program, defined in Figure 3, contains nested recursive calls but has no dependency on data in composite types.

```
for3(i3,step,n) = if(i3>=n) then step else for3(i3+1,step+1,n)
for2(i2,step,n) = if(i2>=n) then step else for2(i2+1,for3(0,step+2,n),n)
for1(i1,step,n) = if(i1>=n) then step else for1(i1+1,for2(0,step,n),n)
tmulta(step,n) = for1(0,step,n)
P(step,n1) = c(step=0)*c(n1=n)
```

**Fig. 3.** The intermediate program containing also the parameterized probability distribution. The parameter  $n$  can obtain only one value.

The nested calls create argument development functions that depend on function calls. These are transformed into a simple form and then removed. The introduced products are over-approximated, but due to the form of the condition the result is precise. The output program computes a single value distribution (when specialized with the size of the matrix). It is given in Figure 4 along with an array describing a subset of specializations of the output program with respect to a value of  $n$ .

	n	program
Ptmulta(out) =	1	Ptmulta(out) = c(out=3)
c(3=<out/(n*n))*	2	Ptmulta(out) = c(out=16)
c(1=<n)*	3	Ptmulta(out) = c(out=45)
c(out/n*n=2+n)*1	4	Ptmulta(out) = c(out=96)
	...	...

**Fig. 4.** The general output probability program (left) and the program specialized with the value of  $n$  (right).

**Adding parameterized distributions.** This example is a recursive program computing the addition of two numbers; the input program and the input probability distribution can be seen in Figure 5. The output depends on both increasing and decreasing values. In this example, we use a parameter  $n$  as the upper limit of a range of input values. The input distribution describes two independent variables, each having a uniform distribution from 1 to  $n$ .

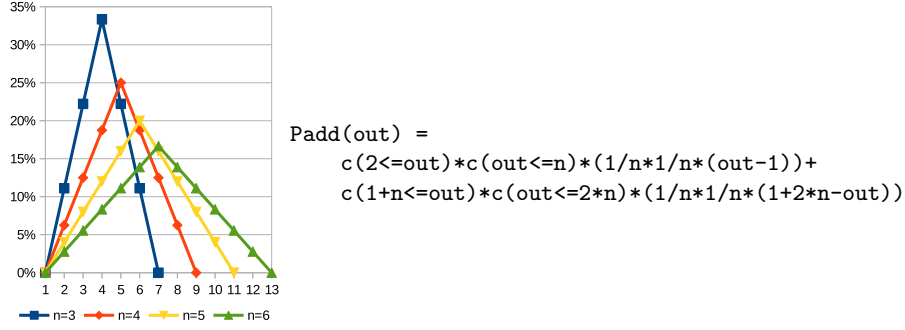
```

add(x,y) = if x<0 then y else add(x-1,y+1)
P(x) = c(1=<x)*c(x=<n)*1/n
Pxy(x,y) = P(x)*P(y)

```

**Fig. 5.** The intermediate program containing both the function `add` and the input probability distribution. Here, the parameter `n` is used to describe a range.

The analysis gives a precise probability distribution and computes a triangular distribution (or pyramid shaped distribution). The output probability program is shown in Figure 6 along with a graph depicting the pyramid shaped output probability distributions for different initializations of `n`. The lower bound on `out` arises from the input probability distribution and not from the condition. The upper bound  $2*n$  of the analysis result shows that the output depends on both input variables, despite that one is increasing and the other is decreasing.



**Fig. 6.** The general output program and the graphs for the output probability distribution with `n` set to 3, 4, 5, and 6, respectively.

**Adding 4 independent variables.** The program `sum4` adds four variables and was presented by Monniaux [23]. Certain over-approximations were applied so as to obtain a safe and simplified result.

The program is recursive and in this example we use independent input variables each uniformly distributed input from 1 to 6, as described in Figure 7.

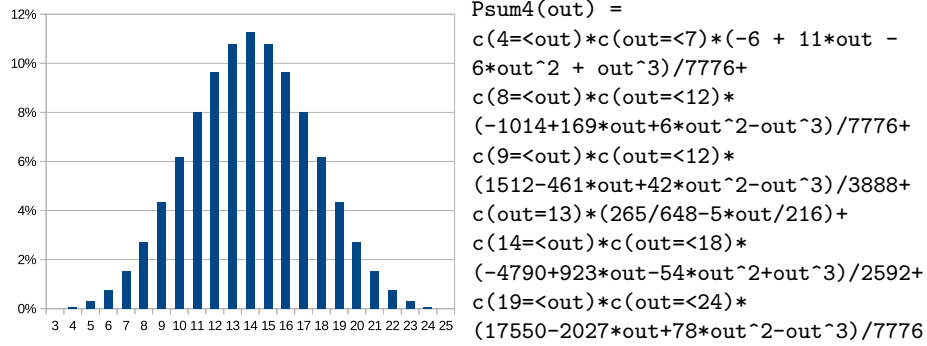
```

add(x,y) = if x=0 then y else add(x-1,y+1)
sum4(x,y,z,w) = add(x,add(y,add(z,w)))
tsum4(x,y,z,w) = sum4(x,y,z,w)
P(x) = c(1=<x)*c(x=<6)*1/6
Pxyzw(x,y,z,w) = P(x)*P(y)*P(z)*P(w)

```

**Fig. 7.** Intermediate program.

Despite the ranges and their associated value are not symmetric, the resulting program computes a precise and perfectly symmetric probability distribution as shown in Figure 8. The difference in the ranges comes (among other things) from the range dividing rules, as they do not divide the range symmetrically. As expected from the central limit theorem of probability theory, the resulting probability program describes a distribution that has similarities with a normal distribution.



**Fig. 8.** The output program and graph for its computed probability distribution for out from 3 to 25.

**Monty Hall.** The Monty Hall problem is often used to exemplify how gained knowledge influences probabilities (conditional probability). In this problem, there are three closed doors; one hiding a price and two that are empty. The doors have an equal chance of hiding the price. There is a contestant, who should choose one of the doors, then the game host will open an empty door and the contestant can either stick with the first choice or can change to the other unopened door. The problem lies in showing whether the best winning-strategy is to stick with the first choice or to switch to the other?

If the strategy is to stick with the first choice and that door has a price then the contestant has won. If the contestant changes door he/she only loses if the first choice was the door hiding the price; if the first choice was an empty door, then the game host would open the other empty door leaving only the price door for a second choice.

The program `monty` models the two strategies; if the strategy variable is 1 then the strategy is to change the door, and otherwise the strategy is to stick with the first choice. The program takes as input the contestant's first guess, the door hiding the price, the empty door which is not opened by the game host and the strategy the contestant uses.

Let us assume the contestant has an equal chance of choosing each of the doors. The input variables `guess`, `price`, and `empty` models the first choice, the price door and the empty door which is left after the game host has opened an empty door. All three doors have a value between 1 and 3, and the empty door cannot be the same as the price door. We have parameterized the strategy with a weight `p` between the two, such that when `p = 1` then the strategy is to always change door, and when `p=0` the strategy is to always keep the first choice (e.g. letting `p = 0.75` we change doors in 3/4 cases and 1/4 we keep the first door). Such a parameterization allows us to execute the analysis once and use the lighter closed form result for that calculation instead. In a problem where the winning-probability of a strategy is dependent on the other input, such input could be used for optimizing the choice of strategy. The program `monty` and the parameterized input probability distribution can be seen in Figure 9.

```

monty(guess,price,empty,strategy)=
  if strategy = 0
    then finalGuess(guess,price)
    else change(guess,price,empty)

finalGuess(guess,price)=
  if price=guess then 1 else 0

change(guess,price,empty)=
  if price=guess
    then finalGuess(empty,price)
    else finalGuess(price,price)

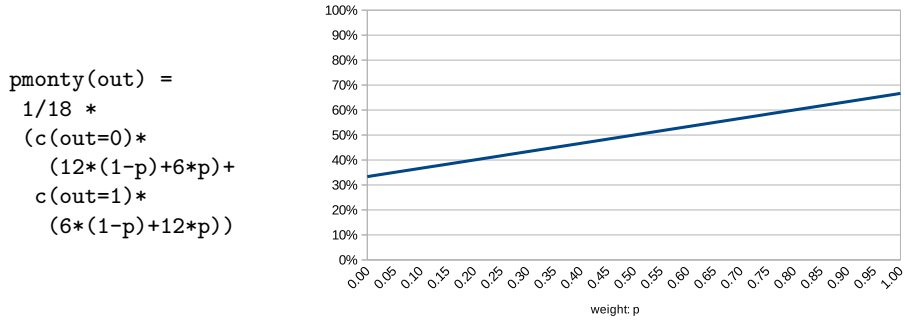
Pin(guess,price,empty,strategy) =
  1/18*c(1=<guess)*c(guess=<3)
    *c(1=<price)*c(price=<3)
    *c(1=<empty)*c(empty=<3)
    *c(not(price = empty))
    *Pstrat(strategy)

Pstrat(strategy) =
  p*c(1 = strategy)
  + (1-p)*c(0 = strategy)

```

**Fig. 9.** The program `monty` models the event flow depending on the chosen strategy; if the strategy is 0 then the contestant keeps the first door and if it is 1 then the contestant changes his mind. There are three doors and the input of `monty` describes the contestants first guess, the door hiding the price, the empty door which is not opened by the game host (and is different from the price door) and the strategy of the contestant. If the final choice hides the price then the program returns 1 and otherwise 0. The probability of the strategy is an expression parameterized with a weight,  $p$  between the two strategies instead of executing the analysis twice.

The analysis was capable of handling the program correctly and the result can be seen in Figure 10.



**Fig. 10.** The probability of winning the Monty Hall as a function of the weight given to change-strategy. The probabilistic output analysis reveals that the best weight between the keep strategy and the change strategy is to always use change strategy.

The probabilities  $1/3$  and  $2/3$  does not occur directly in the output probability program but are found in the constants 6, 12 and  $1/18$ .

**Adding dependent non-uniform variables.** A function call may have interdependent and non-uniform arguments, and in this example, we demonstrate that the analysis can handle such function calls. We focus on the dependencies, analyze a simple add program and discuss the limits of the interdependencies. The program also shows that interdependencies quickly lead to the occurrence of integer division in the output

The input arguments are interdependent; the second argument is always less than or equal to the value of the first argument. The joint distribution depends only on the value of the first argument resulting in a skewed probability distribution. The probability program is defined in Figure 11.

```

Pxy(x,y) =  c(1=<y)*c(y=<3) *
              c(1=<x)*c(x=<y) * x/10
add(x,y) =  x+z

Padd(out) =
  c(2 =< out)*c(out=< 3) * 1/20 * out%2 * (1 + out%2) +
  c(4 =< out)*c(out=< 6) * -(1/20)*(-4+out-out%2)*(-3+out+out%2)

```

**Fig. 11.** An input program, `add`, its skewed joint distribution, `Pxy`, and the closed form probability program, `Padd`, produced by the analysis. The integer division is noted by a “%”.

The create rule generates nested summations, and removing such inner summations imply that their values must be expressed using the variables of the outer summations or the input variable (ie. `out`). Comparing the result from this experiment with the output probability distribution for addition of two random variables in Figure 6 indicates that integer division is a special case arising from dependent input. The following interesting expressions are extracted during analysis execution, and they shows how the integer division arises from the dependency of input. The first expressions is the result from the create rule and the last expression is the result after removal of the inner `y`-summation.

$$\begin{aligned}
P_{add}(out) = & \text{sum}(x; \text{sum}(y; c(out =_i x +_i y) \times_q \\
& c(1 \leq_i x) \times_q c(x \leq_i y) \times_q c(1 \leq_i y) \times_q c(y \leq_i 3) \times_q (i2r(x) /^q i2r(10)))) = \\
& \text{sum}(x; c(2 \leq_i out) \times_q c(out \leq_i 3) \times_q c(1 \leq_i x) \times_q \\
& c(2 \times_i x \leq_i out) \times_q (i2r(x) /^q i2r(10))) +_q \\
& \text{sum}(x; c(4 \leq_i out) \times_q c(out \leq_i 3 +_i x) \times_q c(2 \times_i x \leq_i out) \times_q (i2r(x) /^q i2r(10)))
\end{aligned}$$

In the last expression, there are two summations, each leading to its own part in the resulting program. Looking closely at each summation, we see that they share the upper limit for `x`, `c(2 ×i x ≤i out)`, which currently contains an integer multiplication and when solved with respect to `x` contains the integer division. In the final result, the second part of the expression has an upper limit for `out`, `c(out ≤i 6)` which is a constraint that the summation-removal-rule introduces to ensure that the lower limit of the summation (i.e. `out −i 3`) is less than or equal to the upper limit (i.e. `out %i 2`).

The original probability  $(i2r(x) /^q i2r(10))$  occurs directly in the summations, and this indicates a limit of this implementation and approach. To be able to handle a probability, the rewrite rules for summations must transform summations over the probability expression. There are limits to which series the system currently can transform, Sum of reciprocals (e.g.  $\sum_{k=1}^n \frac{1}{k}$ ) known as



harmonic series or variations thereof such as generalized harmonic series are currently not implemented. The current analysis is limited to finite summations of at least order of 1, but a closer integration with Mathematica that exploits more of Mathematica's rewriting mechanisms should be able to handle such series.

## 7 Related works

Probabilistic analysis is related to the analysis of probabilistic programs. Probabilistic analysis analyze programs with a normal semantics where the input variables are interpreted over probability distributions. Analysis of probabilistic programs analyze programs with probabilistic semantics where the values of the input variables are unknown (e.g. flow analysis [25]).

In probabilistic analysis, it is important to determine how variables depend on each other, but already in 1976 Denning proposed a flow analysis for revealing whether variables depend on each other [8]. This was presented in the field of secure flow analysis. Denning introduced a lattice-based analysis where she, given the name of a variable, that should be kept secret, deducted which other variables those should be kept secret in order to avoid leaking information. In 1996, Denning's method was refined by Volpano et al. into a type system and for the first time, it was proven sound [33].

Reasoning about probabilistic semantics is a closely related area to probabilistic analysis, as they both work with nested probabilistic influence. The probabilistic analysis work on standard semantic and analyze it using input probability distributions, where a probabilistic semantics allow for random assignments and probabilistic choices [20] and is normally analyzed using an expanded classical analysis or verification method [6].

Probabilistic model checking is an automated technique for formally verifying quantitative properties for systems with probabilistic behaviors. It is mainly focused on Markov decision processes, which can model both stochastic and non-deterministic behavior [13, 21]. It differs from probabilistic analysis as it assumes the Markov property.

In 2000, Monniaux applied abstract interpretation to programs with probabilistic semantics and gained safe bounds for worst-case analysis [23]. Pierro et al. introduce a linear mapping structure, a Moore-Penrose pseudo-inverse, instead of a Galois connection. They use the linear structures to compare 'closeness' of approximations as an expression using the average approximation error. Pierro et al. further explores using probabilistic abstract interpretation to calculate the average-case analysis [24]. In 2012, Cousot and Monerau gave a general probabilistic abstraction framework [6] and stated, in section 5.3, that Pierro et al.'s method and many other abstraction methods can be expressed in this new framework.

When analyzing probabilities the main challenge is to maintain the dependencies throughout the program. Schellekens defines this as *Randomness preservation* [30] (or random bag preservation) which in his (and Gao's [14]) case enables tracking of certain data structures and their distributions. They use special data

structures as they find these suitable to derive the average number of basic operations. In another approach [34, 26], tests in programs has been assumed to be independent of previous history, also known as the Markov property (the probability of true is fixed). As Wegbreit remarked, this is true only for some programs (e.g. linear search for repeating lists) and others, this is not the case (linear search for non-repeating lists). The Markov property is the foundation in Markov decision processes which is used in probabilistic model-checking [13]. Cousot et al. presents a probabilistic abstraction framework where they divide the program semantics into probabilistic behavior and (non-)deterministic behavior. They propose handling of tests when it is possible to assume the Markov property, and handle loops by using a probability distribution describing the probability of entering the loop in the  $i$ th iteration. Monniaux proposes another approach for abstracting probabilistic semantics [23]; he first lifts a normal semantics to a probabilistic semantics where random generators are allowed and then uses an abstraction to reach a closed form. Monniaux’s semantic approach uses a backward probabilistic semantics operating on measurable functions. This is closely related to the forward probabilistic semantics proposed earlier by Kozen [20].

An alternative approach to probabilistic analysis is based on symbolic execution of programs with symbolic values [15]. Such techniques can also be used on programs with infinitely many execution paths by limiting the analysis to a finite set of paths at the expense of tightness of probability intervals [29].

## 8 Conclusion

Probabilistic analysis of program has a renewed interest for analyzing programs for energy consumptions. Numerous embedded systems and mobile applications are limited by restricted battery life on the hardware. In this paper, we describe a rewrite system that derives a resource probability distribution for programs given distributions of the input. It can analyze programs in subset of C where we have known distribution of input variables. From the original program we create a probability distribution program, where we remove calls to original functions and transform it into closed form. We have presented the transformation rules for each step and outlined the implementation of the system. We discuss over-approximating rules and their influence on the accuracy of the output probability and show that our analysis improves on related analysis in the literature.

## References

- [1] A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault, and S. Putot. Static analysis of programs with imprecise probabilistic inputs. In *In Verified Software: Theories, Tools, Experiments*, pages 22–47. Springer Berlin Heidelberg., 2014.
- [2] M. Bauer. Approximations for decision making in the Dempster-Shafer theory of evidence. In E. Horvitz and F. V. Jensen, editors, *UAI*, pages 73–80. Morgan Kaufmann, 1996.
- [3] D. Berleant and H. Cheng. A software tool for automatically verified operations on intervals and probability distributions. *Reliable Computing*, 4(1):71–82, 1998.

- [4] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. A generalization of p-boxes to affine arithmetic. *Computing*, 94(2-4):189–201, 2012.
- [5] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-Garcia, and G. Puebla. The ciao prolog system. *Reference Manual. The Ciao System Documentation Series—TR CLIP3/97.1*, School of Computer Science, Technical University of Madrid (UPM), 95:96, 1997.
- [6] P. Cousot and M. Monerau. Probabilistic abstract interpretation. In H. Seidl, editor, *ESOP*, volume 7211 of *LNCS*, pages 169–193. Springer, 2012.
- [7] Saumya K Debray, P López García, Manuel Hermenegildo, and N-W Lin. Estimating the computational cost of logic programs. In *Static Analysis*, pages 255–265. Springer, 1994.
- [8] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
- [9] S. Destercke and D. Dubois. The role of generalised p-boxes in imprecise probability models. In *6th International Symposium on Imprecise Probability: Theories and Applications*, 2009.
- [10] S. Ferson. Model uncertainty in risk analysis. Tech. report, Centre de Recherches de Royallieu, Université de Technologie de Compiègne, 2014.
- [11] S. Ferson, V. Kreinovich, L. Ginzburg, D. S. Myers, and K. Sentz. Constructing probability boxes and Dempster-Shafer structures. Sand2002-4015, Sandia National Laboratories, 2002.
- [12] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithm. *Theor. Comput. Sci.*, 79(1):37–109, 1991.
- [13] V. Forejt, M. Z. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In M. Bernardo and V. Issarny, editors, *SFM*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.
- [14] A. Gao. *Modular average case analysis: Language implementation and extension*. Ph.d. thesis, University College Cork, 2013.
- [15] J. Geldenhuys, M. B Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176. ACM, 2012.
- [16] J. Gordon and E. H. Shortliffe. The Dempster-Shafer theory of evidence. In *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, page 21 pp, 1984.
- [17] X. Guo, M. Boubekur, J. McEnery, and D. Hickey. ACET based scheduling of soft real-time systems: An approach to optimise resource budgeting. *International Journal of Computers and Communications*, 1(1):82–86, 2007.
- [18] R. U. Kay. Fundamentals of the Dempster-Shafer theory and its applications to system safety and reliability modelling. In *RTA*, pages 173–185, 2007.
- [19] S. Kerrison and K. Eder. Energy modelling and optimisation of software for a hardware multi-threaded embedded microprocessor. *University of Bristol, Bristol, Tech. Rep*, 2013.
- [20] D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.
- [21] M. Kwiatkowska, G. Norman, and D. Parker. Advances and challenges of probabilistic model checking. In *48th Annual Allerton Conference on Communication, Control, and Computing*, pages 1691–1698. IEEE, September 2010.
- [22] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder. Energy consumption analysis of programs based on xmos isa level models. In *23rd International Symposium on Logic-Based Program*

- Synthesis and Transformation, LOPSTR*, volume 8901 of *LNCS*, pages 72–90, 2013.
- [23] D. Monniaux. Abstract interpretation of probabilistic semantics. In Jens Palsberg, editor, *SAS*, volume 1824 of *LNCS*, pages 322–339. Springer, 2000.
  - [24] A. Di Pierro, C. Hankin, and H. Wiklicky. Abstract interpretation for worst and average case analysis. In T. W. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation*, volume 4444 of *LNCS*, pages 160–174. Springer, 2006.
  - [25] A. Di Pierro, H. Wiklicky, G. Puppis, and T. Villa. Probabilistic data flow analysis: a linear equational approach. In *Proceedings Fourth International Symposium*, volume 119, pages 150–165. Open Publishing Association, 2013.
  - [26] H. Soza Pollman, M. Carro, and P. Lopez Garcia. Probabilistic cost analysis of logic programs: A first case study. *INGENIARE - Revista Chilena de Ingeniera*, 17(2):195–204, 2009.
  - [27] M. Rosendahl. Automatic complexity analysis. In *FPCA*, pages 144–156, 1989.
  - [28] M. Rosendahl and M. H. Kirkeby. Probabilistic output analysis by program manipulation. In *Quantitative Aspects of Programming Languages*, EPTCS, 2015.
  - [29] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *In Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 447–458. ACM., June 2013.
  - [30] M. P. Schellekens. *A modular calculus for the average cost of data structuring*. Springer, 2008.
  - [31] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.
  - [32] A. Uwimbabazi. Extended probabilistic symbolic execution. Master’s thesis, University of Stellenbosch, 2013.
  - [33] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
  - [34] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.
  - [35] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
  - [36] A. Wierman, L. L. H. Andrew, and A. Tang. Stochastic analysis of power-aware scheduling. In *Proceedings of Allerton Conference on Communication, Control and Computing*. Urbana-Champaign, IL, 2008.
  - [37] N. Wilson. Algorithms for Dempster-Shafer theory. In *Handbook of defeasible reasoning and uncertainty management systems*, pages 421–475. Springer Netherlands, 2000.
  - [38] S. Wolfram. The Mathematica book. *Cambridge University Press and Wolfram Research, Inc., New York, NY, USA and*, 100:61820–7237, 2000.
  - [39] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis*, pages 280–297. Springer, 2011.